Pc-pelin kehitys ja julkaisu

Henri Lahti

Opinnäytetyö
Joulukuu 2014
Tietojenkäsittely

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely

LAHTI, HENRI:
Pc-pelin kehitys ja julkaisu

Opinnäytetyö 42 sivua, joista liitteitä 1 sivua
Joulukuu 2014

Pelien markkinat kasvavat ympäri maailmaa jatkuvasti ja nykyisten työkalujen avulla pelien tekeminen onnistuu hyvinkin pienillä resursseilla. Tämän opinnäytetyön tavoitteena oli antaa tilaajana toimivalle RefleKT Media -yritykselle yleiskuva pelinkehityksestä, jonka avulla yritys pystyy monipuolistamaan tietotaitoaan ja parhaassa tapauksessa auttaisi hankkimaan uusia projekteja pelien parissa. Tarkoituksena oli toteuttaa yhdessä tilaajan edustajan kanssa PC- ja Mac-alustoilla pelattava peli sekä tutkia, kuinka sen julkaiseminen onnistuisi digitaalisia kanavia pitkin.

Opinnäytetyössä toteutettiin Run the Gauntlet -nimeä kantava peli Unity-pelimoottoria käyttäen. Työssä käsitellään teoriatasolla pelinkehityksessä tarvittavia työkaluja sekä käydään Unityn toimintaa pääpiirteissään läpi. Pelin toteutuksesta kertovassa osiossa käsitellään pelin suunnittelua sekä muutamia osa-alueita pelin tekemisestä Unitylla. Toteutusosio painottuu pelin toiminnallisuuksien ohjelmoimiseen C#-kielellä, joten lukijalta oletetaan jonkin verran kokemusta ohjelmoinnista. Lopuksi työssä käydään teoreettisesti läpi, miten tietokonepelin julkaisu tapahtuisi digitaalisesti.

Opinnäytetyön lopputuloksena on rungoltaan toimiva ja helposti laajennettavissa oleva tietokonepeli, joka kuitenkin vaatisi parannuksia etenkin audiovisuaalisesti, jotta se saataisiin julkaistavaan kuntoon. Tilaajalle projekti antoi varsin kattavan yleiskatsauksen pelin kehittämiseen vaadittavista resursseista ja taidoista. Jatkokehittämällä nykyistä versiota tilaaja saa itselleen hyvän referenssinäytteen, jonka avulla ovet pelialalle voisivat hyvinkin aueta.

Asiasanat: unity, peliohjelmointi, tietokonepeli, digitaalinen julkaisu, pelinkehitys

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems

LAHTI, HENRI:
Implementation of PC Game and Publishing

Bachelor's thesis 42 pages, appendices 1 pages
December 2014

Game markets are growing constantly all over the world. Nowadays developing a game has become more achievable even for small teams because of advanced tools and distribution does not need massive investment anymore. The objective of this thesis was to provide perspective on game development for the commissioner company called RefleKT Media and possible ways to extend company operations to include game projects. The purpose of this project was to implement a computer game together with commissioner's agent and research how it can be published through digital distribution channels.

During this project tower defence game called Run the Gauntlet was implemented by using Unity game engine. Theoretical section of this thesis explores tools which can be used for game developing and how completed game can be published using digital channels. The main focus of tools section is on Unity because it was a big part of this project. In the implementation, there is section about designing of Run the Gauntlet and developing the game. The implementation part is biased towards game programming so it was presumed that the reader have basic knowledge about programming.

The result is a playable game that can be easily expanded with new content but it still needs some upgrades especially for graphics and sounds before it can be released. For the commissioner the project provided wide view of what the game development really is. In the future this game might give good reference for RefleKT Media to expand their actions in game industry projects.

Key words: unity, game programming, computer game, digital distribution, game development

SISÄLLYS

1 JOHDANTO

Suomen peliala on elänyt viime vuosina vahvassa nousujohteessa, joka näkyy muun muassa yritysten määrän sekä alan liikevaihdon kasvuna. Vuosituhannen alussa Suomessa oli noin kymmenkunta pelialan toimijaa (Hiltunen, K. Latva, S. Kaleva, P. 2013, 16) kun tätä nykyä peleihin keskittyneiden yritysten määrä on ylittänyt yli kahden sadan toimijan rajapyykin (Neogames: Alan toimijat 2014). Samalla alan liikevaihto on kasvanut muutamassa vuodessa alle 100 miljoonasta jopa 800 miljoonaan euroon vuodessa (Neogames: Tietoa toimialasta 2014). Vaikka monet pelejä kehittävistä yrityksistä ovatkin hyvin pieniä, mahtuu mukaan myös Rovion tai Supercellin kaltaisia menestystarinoita, jotka ovatkin osaltaan vaikuttaneet suuresti suomalaisen pelinkehitysbisneksen nousuun omalla esimerkillään. Näille firmoille yhteistä on kuitenkin se, että molemmat tekevät pääsääntöisesti mobiilipelejä. Rovion Angry Birds on julkaistu myös muille alustoille, mutta kyseisen brändin menestyksen voidaan katsoa alkaneen nimenomaan vuonna 2009 julkaistusta mobiiliversiosta (Rigney 2010).

Suomalaisten kehittämillä mobiilipeleillä menee lujaa, mutta muiden alustojen pelit eivät ole saavuttaneet aivan yhtä suurta suosiota maailmalla. Etenkin puhtaasti tietokoneelle kehitettyjä merkittäviä julkaisuja ei viime vuosina ole ollut kuin kourallinen, joka on ajanut monet Suomen mittapuulla suuret firmat, kuten Remedy ja Housemarque, siirtymään konsoleiden pariin. Toki kyseinen ilmiö ei koske vain suomalaisia pelitaloja, vaan kyseessä on hyvin globaali ilmiö. Tätä voidaan selittää puhtaasti pelibisneksen jatkuvasti kasvavilla tuotantoarvoilla, jolloin keskiverto A-luokan pelin tulee myydä jopa miljoonia kappaleita, kattaakseen kehitykseen aikana kertyneet kulut. Konsoleiden lähestyessä tekniikaltaan tietokoneita, onkin siis järkevää tehdä niin sanottu monialustajulkaisu, jolloin sama tuote saadaan pienin muutoksin käännettyä usealle eri alustalle ja samalla suuremman kuluttajakunnan saataville. Monialustajulkaisu voi tulla kuitenkin hyvin kalliiksi, sillä konsolijulkaisuun vaadittavat lisenssimaksut ja mahdollisen pelilevyn painattamisesta ynnä muusta aiheutuvat kulut voivat olla kuitenkin pienelle firmalle liian suuret. Pc-pelin puolestaan voi kehittää nykypäivänä hyvinkin pienillä resursseilla, sillä tarjolla on lukuisia järkevän hintaisia työkaluja ja julkaisun voi hoitaa täysin digitaalisten kauppojen avulla, jolloin julkaisukulut voivat olla hyvin minimaaliset. Tämä opinnäytetyö käsitteleekin millä keinoin voidaan toteuttaa yksinomaan tietokoneelle tarkoitettu peli ja keinoja sen julkaisemiseen.

2 TYÖN TAUSTA

2.1 Tavoite ja tarkoitus

Tämän opinnäytetyön tilaajana toimii yritys nimeltään RefleKT Media. Työn tavoitteena on antaa tilaajalle yleiskuva siitä, kuinka paljon aikaa ja resursseja pelinkehitys vaatii sekä luoda mahdollinen ponnahduslauta pelialan pariin ja tätä kautta hankkia uusia projekteja pelien parissa.

Tarkoituksena on yhdessä tilaajan kanssa toteuttaa PC/Mac-alustalla pelattava peli Unity-pelimoottoria käyttäen, jota olisi tulevaisuudessa helppo laajentaa päivityksillä. Samalla tarkoitus on selvittää miten ja minkä digitaalisen palvelun kautta pelin mahdollinen julkaisu voitaisiin toteuttaa.

2.2 Työn toteutus

Työn aikana kehitetään tornipuolustuspeli ja kehitystiimiin kuuluu itseni lisäksi tilaajan edustaja Karlo Tuominen, joka vastaa ohjauksesta, sekä luo pelin grafiikat ja toteuttaa käyttöliittymän. Oma roolini on osallistua pelin suunnitteluun ja kenttien rakenteluun, mutta pääasiallinen vastuualueeni on pelin eri toimintojen ohjelmoiminen.

Työ alkaa suunnittelulla, jonka aikana päätetään käytettävät työkalut, sekä pelin pääpiirteet. Toteutuksessa hyödynnetään ketteriä menetelmiä, joten alustavia suunnitelmia voidaan kehityksen aikana muokata joustavasti.

Projekti viedään läpi niin sanotulla nollabudjetilla, joten käytettävien kehitystyökalujen valinnassa tähdätään ilmaisuuteen.

3 TYÖKALUT KEHITYKSEEN

3.1 Pelimoottorit

Pelin kehittämiseen on tarjolla monia vaihtoehtoja: kaiken voi tehdä itse tai vaihtoehtoisesti käyttää olemassa olevia pelimoottoreita, jotka ovat ohjelmistokehyksiä pelien tekoon. Ne sisältävät usein paljon valmiita komponentteja, joita voidaan hyödyntää kehityksessä, kuten luokkakirjaston ohjelmointiin, fysiikkamoottorin, käyttövalmiita syötteentunnistimia kontrolleja varten, tai vaikkapa työkalun peliobjektien reitinhaun toteuttamiseen. (Ward 2008.) Pelimoottoreiden etuna onkin, että kehittäjät voivat keskittyä suunnitteluun ja toteutukseen sen sijaan, että pelkästään teknisen pohjan luomiseen käytettäisiin pahimmillaan useita vuosia.

Valmiita pelimoottoreita on olemassa hyvin erilaisia, joista osa antaa kehittäjälle nipun työkaluja, joilla saadaan aikaan valmis tuote. Jotkut taas toimivat vain käyttöliittymänä, johon kehittäjä voi itse liittää tarvitsemansa määrän kaupallisia, tai open source -pohjaisia lisävarusteita. (Ward 2008.) Pelimoottoria valittaessa onkin huomioitava kehitysryhmän taitotaso sekä pohdittava, minkälaisia ominaisuuksia pelimoottorin tulisi sisältää, sillä väärin tehty valinta voi johtaa umpikujaan, jossa pelimoottori ei taivukaan haluttuun lopputulokseen. Tällöin edessä voi olla paljon turhaa työtä, kun koko projekti joudutaan siirtämään uudelle kehitysalustalle.

Valinnan tekeminen ei aina ole kovin helppoa, sillä nykyaikaiset pelimoottorit tukevat hyvin paljon samoja ominaisuuksia toistensa kanssa. Tällaisia ovat esimerkiksi monen ytimen tuki, mahdollisuus kääntää sama versio tuotoksesta usealle eri alustalle, sekä tuki useammalle kuin yhdelle ohjelmointikielelle. Samankaltaisuuksien takia onkin tarpeellista tutkia, tukeeko pelimoottori muita projektissa käytettäviä sovelluksia, kuten esimerkiksi 3D-mallinnusohjelma. Toinen helpottava asia on lueskella läpi moottorin dokumentaatiota, joka löytyy monesti kokonaisuudessaan valmistajan sivuilta. Dokumentaatio sisältää yleensä täydellisen listauksen moottorin teknisistä ominaisuuksista ja moottorilla tehtävien pelien lisensoinnista, joissa voi olla rahallisesti suuriakin eroja. Aloittelevalle kehittäjälle valintaa helpottava tekijä on myös moottorin ominaisuuksista

tehdyt tutoriaalit, joiden avulla uuden tekniikan opetteleminen on huomattavasti helpompaa.

3.1.1 **GameMaker**

GameMaker on YoYo Gamesin julkaisema pelimoottori, joka mahdollistaa pelien kehittämisen tietokoneille sekä mobiililaitteille. Mac-laitteita käyttävien kehittäjien kannattaa huomioida, että pelimoottorin uusin versio (8.1) toimii tällä hetkellä vain Windows-alustalla. GameMaker eroaa kilpailijoistaan vahvalla suuntautumisella 2D-peleihin, vaikka sekin sisältää mahdollisuuden kolmiulotteisuuteen, mutta ei pääse siinä aivan kilpailevien moottorien tasolle. GameMakerin ilmaisversio mahdollistaa pelien tekemisen ainoastaan Windows-käyttöjärjestelmälle. Muiden alustojen kehityslisenssit vaativat lisäksi Professional-version omistamisen sekä alustakohtaisen lisenssin. (GameMaker: Studio 2014.) Pelien ohjelmointi suoritetaan sisäänrakennetun editorin avulla, käyttäen GameMakerin omaa Game Maker Language (GML) -skriptauskieltä (GameMaker Documentation 2014).

3.1.2 **Unreal Engine**

Unreal Engine on yksi tämän hetken tehokkaimmista pelimoottoreista ja vastikään julkaistu moottorin 4. versio kykenee ainakin kuvien perusteella luomaan graafisesti upean näköisiä pelejä. Pelimoottorilla on mahdollista toteuttaa projekteja lähes jokaiselle tämän hetken pelialustoista ja sillä voidaan tehdä niin kaksi- kuin kolmiulotteisia pelejä. Moottorin edellisessä versiossa ohjelmointi suoritetaan käyttäen Unrealscript-kieltä, joka on toiminnaltaan hyvin samankaltainen C/C++- sekä Java-kielien kanssa (Unreal Script Reference 2012). Unrealscript ei nimestään huolimatta ole puhdas skriptauskieli, sillä esimerkiksi Unityn käyttämistä Javascriptista ja C#:sta poiketen, se tarvitsee erikseen kääntää, jotta tehdyt muutokset tulevat voimaan (Doran 2013, 243). Moottorin neljäs versio puolestaan käyttää ohjelmointikielenä C++:aa (Unreal Documentation: Programming Basics 2014), jonka oppiminen voi olla ohjelmoijalle aluksi hieman haastavampaa. Rekisteröitymällä unreal-kehittäjäksi saa 19 euron kuukausimaksua vastaan

kaiken muun lisäksi pääsyn moottorin lähdekoodiin, joten Unreal engine on hyvin sovitettavissa kehittäjän tarpeisiin, eikä sen toiminta ole näkymättömissä. Kuukausimaksun lisäksi Epic games perii 5% osuuden mahdollisista tuotoista sen jälkeen, kun pelin kokonaistuotto ylittää kolmen tuhannen dollarin rajapyykin. (Unreal Engine 2014.) Aikaisemmat versiot ovat olleet erittäin käytettyjä myös isojen AAA -nimikkeiden kehittämisessä, kuten miljoonia myyneissä pelisarjoissa 2K Gamesin Bioshock, sekä Warner Brossin Batman: Arkham pelit (Unreal Engine Showcase 2014).

3.1.3 **Unity**

Unity on Unity Technologiesin kehittämä pelimoottori, jolla voidaan kehittää pelejä tietokoneille, selaimeen, mobiililaitteille, sekä osalle nykykonsoleista. Sen avulla voidaan toteuttaa sekä kaksi- että kolmiulotteisia pelejä ja pelimoottoriin kuuluu laaja valikoima erilaisia työkaluja kehitykseen. Unity ei Unreal Enginen tavoin sisällä mahdollisuutta muokata moottorin lähdekoodia, joten sen toimintaan ei voi tästä johtuen vaikuttaa yhtä laajasti. (Unity3D 2014.) Unitysta kerron laajemmin myöhemmin tulevassa luvussa.

3.2 Muut ohjelmat

Kehitettiinpä peliä sitten natiivisti tai pelimoottorin avulla, tarvitaan toteuttamiseen aina hyvin paljon muutakin, jota pelimoottorit yksistään eivät mahdollista. Yleensä pelin kehitykseen tarvitaan jonkinlainen sovellus hahmomallien ja erilaisten materiaalien luomiseen, johon soveltuu käytännössä mikä tahansa piirto-ohjelma kuten Adoben Photoshop, tai vaikka Paint. 3D-pelien kohdalla tarvitaan hieman monimutkaisempi sovellus, joka kykenee kolmiulotteisten mallien, kuten pelihahmon, tai rakennuksen, luomiseen. 3D-mallinnusohjelmia ovat esimerkiksi Cinema 4D, Blender sekä Maya. Piirto-ohjelmat mahdollistavat joidenkin pelimoottorien kohdalla myös pelin kenttäpohjien piirtämisen, joiden avulla pelimoottori kykenee luomaan tasaisesta kuvasta 3D-kentän. Lisäksi pelin äänien tekemiseen on hyvä olla erillinen sovellus, jolla voidaan tarvittaessa äänittää, tai luoda äänikirjaston pohjalta erilaisia audiotiedostoja aina musiikista eri-

laisiin kolahduksiin, tai ampumisääniin ja dialogiin. Etenkin materiaalien ja äänien kohdalla on mahdollista hyödyntää Internetistä löytyviä open source -kirjastoja. Kirjastojen kohdalla kannattaa olla kuitenkin tarkkana lisensoinnin kanssa, ettei myöhemmin tule ongelmia tekijänoikeuksien suhteen.

3.3 Kauppapaikat

Monilta pelimoottorien valmistajilta löytyy kauppapaikka, josta on saatavilla hyödyllisiä apuvälineitä kehitykseen. Nämä nopeuttavat kehitystä, sillä kauppapaikoilta voi löytää vaikkapa animointiin tai hahmojen tekoälyn toteuttamiseen apuvälineitä, jotka mahdollistavat muutamalla klikkauksella toimintoja, joiden ohjelmointiin saataisi muuten kulua päiväkausia. Kaupan apuvälinekirjo on usein laaja ja kattaa kaiken, aina pelihahmoista koodinpätkiin, tai vaikka erilaisiin partikkelijärjestelmiin. Hintaskaala on myös vaihteleva, sillä apuvälineet saattavat olla täysin ilmaisia, tai laajempien pakettien kohdalla ne voivat maksaa jopa useita satoja euroja. Kauppapaikan ei tulisi olla kuitenkaan kovin painava peruste pelimoottoria valittaessa, sillä sieltä saatavat välineet ovat toki hyödyllisiä, mutta eivät mitenkään välttämättömiä pelin kehityksen kannalta.

4 UNITY

Tähän projektiin pelimoottoriksi valikoitui Unity Tecnologiesin alunperin vuonna 2005 lanseeraama Unity3D, joka kantaa tätä nykyä enää nimeä Unity. Unity on edennyt versioon 4.5.4 ja vuosien varrella siihen on lisätty lukuisia ominaisuuksia kuten 2D-ominaisuudet sekä reitinhakutyökalu. Maailmanlaajuisesti kyseinen pelimoottori hallitsee markkinoista 45% ja 47% kehittäjistä on ottanut Unityn työvälineeksi projekteissaan. (Unity: Public Relations 2014). Tämän moottorin valintaa edesauttoivat sen kattava dokumentaatio, tuki Blender 3D-mallinnusohjelmalle, sekä kehittäjäryhmän kokemus kyseisestä pelimoottorista. Lisäksi Unity taipuu hyvin 2.5D-pelin luomiseen, jossa pelimaailma on esitetty 3D:nä, mutta kamera on lukittu 2D-kuvakulmaan.

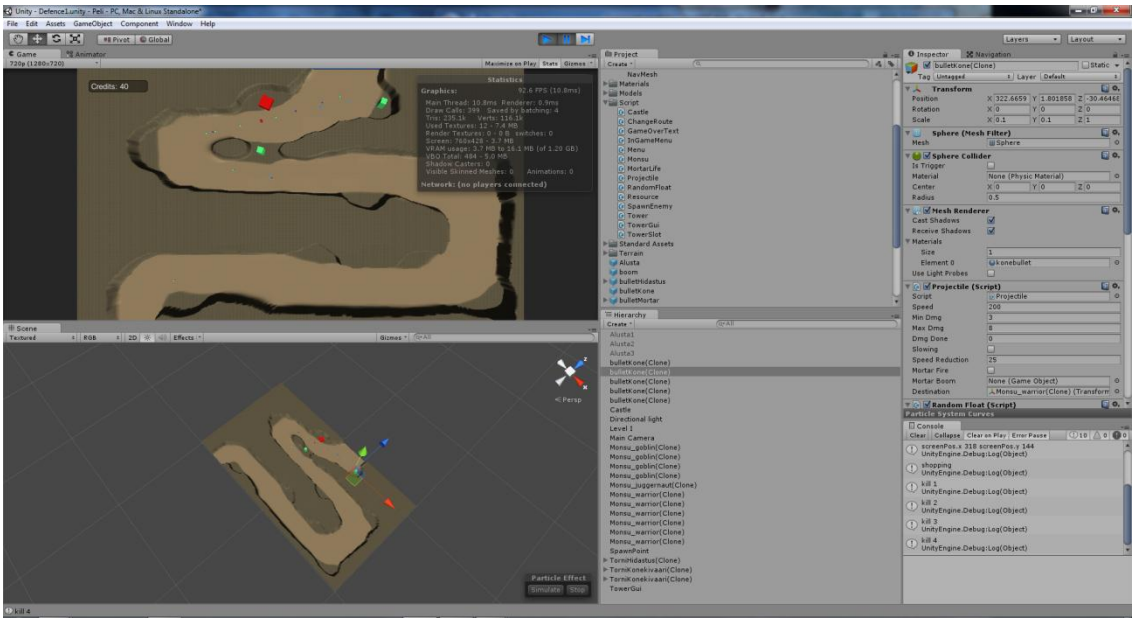
4.1 Yleistä

Unity on työkalujensa puolesta hyvin kattava pelimoottori. Siitä löytyy sisäänrakennettuna muun muassa törmäyksen tunnistus, animointityökalu, erilaisia valaistuskomponentteja sekä fysiikkamoottori. Seuraavassa versiossa, 4.6, pelimoottoriin lisätään myös pelin käyttöliittymän rakennusta helpottava työkalu, jonka avulla GUI (graphical user interface) voidaan luoda tartu ja vedä- tyyliin, kun nykyisessä versiossa se luodaan puhtaasti skriptaamalla (Unity Learn: The new UI 2014). Editorin ulkoasu on myös sovitettavissa melko vapaasti käyttäjän mieltymyksien mukaan. Unity tukee useita eri mallinnusohjelmia, joten omien hahmojen, rakennusten tai materiaalien tuominen pelimoottoriin onnistuu varsin vaivattomasti.

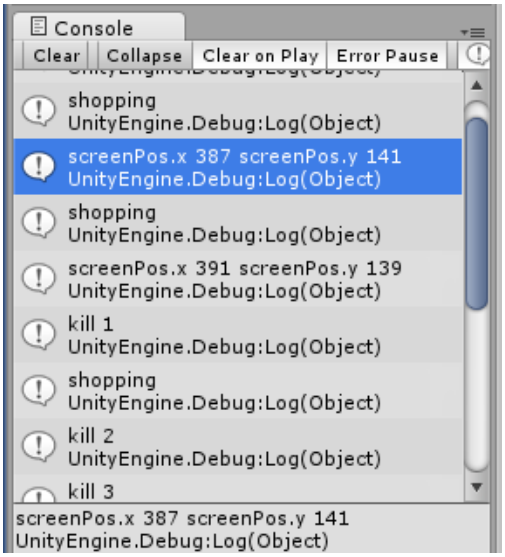
Unitysta on tarjolla ilmaisversio, jota on käytetty tässä opinnäytetyössä, sekä maksullinen Pro -versio, jonka mukana avautuu monia ilmaisversiossa lukittuina olevia graafisia hienouksia, sekä työkalujen ominaisuuksia. Ilmaista versiota voi käyttää myös kaupallisiin projekteihin, mikäli käyttävän instituution liikevaihto on alle 100 000 dollaria. Pro -versiota voi kokeilla kuukauden ajan ilmaiseksi, jonka jälkeen sen voi lunastaa omaksi hieman yli tuhannen euron hintaan, tai noin 60 euron kuukausimaksulla. Jotkut alustat vaativat myös omat lisenssinsä, jotka maksavat saman verran kuin Unityn Pro -lisenssi. (Unity FAQ 2014; Unity: Store 2014.)

4.2 Käyttöliittymä

Pelin tekeminen Unitylla suoritetaan graafisen editorin avulla, jonka kautta voidaan muokata kaikkea muuta paitsi pelin koodia. Käyttäjä voi jakaa editorin haluamansa kokoisiin ikkunoihin, joihin hän voi valita tarvitsemansa työkalut. Yhteen ikkunaan voidaan vetää useita työkaluja, jolloin siihen ilmaantuu uusi välilehti. Kuvassa 1 näkyy itse käyttämäni asettelu. Vasemmassa reunassa on ikkunat pelinäkymästä (Game) ja sen alapuolella kenttäikkuna (Scene). Keskimmäiset ikkunat ovat projektikansiota, josta löytyvät kaikki pelin assetit. Sen alla on auki olevan kentän hierarkia, jossa näkyvät kaikki kenttään sijoitetut assetit. Oikeassa reunassa on ensin kaksi välilehteä sisältävä ikkuna, johon on sijoitettu gameobjectin tiedot näyttävä inspector ja navigaatiotyökalu. Tämän alla on puolestaan konsoli, jossa näkyvät virheilmoitukset, sekä Debug -luokan ilmoitukset pelin ajon aikana (kuva 2). Unityssa on myös monia valmiita asetteluja.

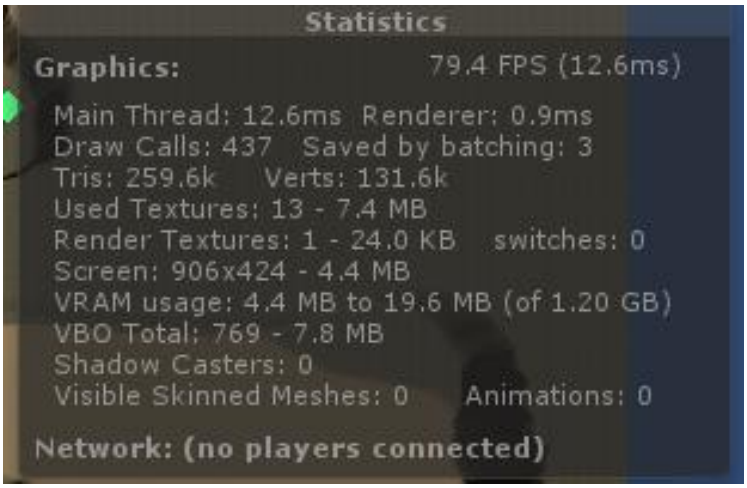


KUVA 1. Kuvakaappaus Unityn editorinäkymästä.

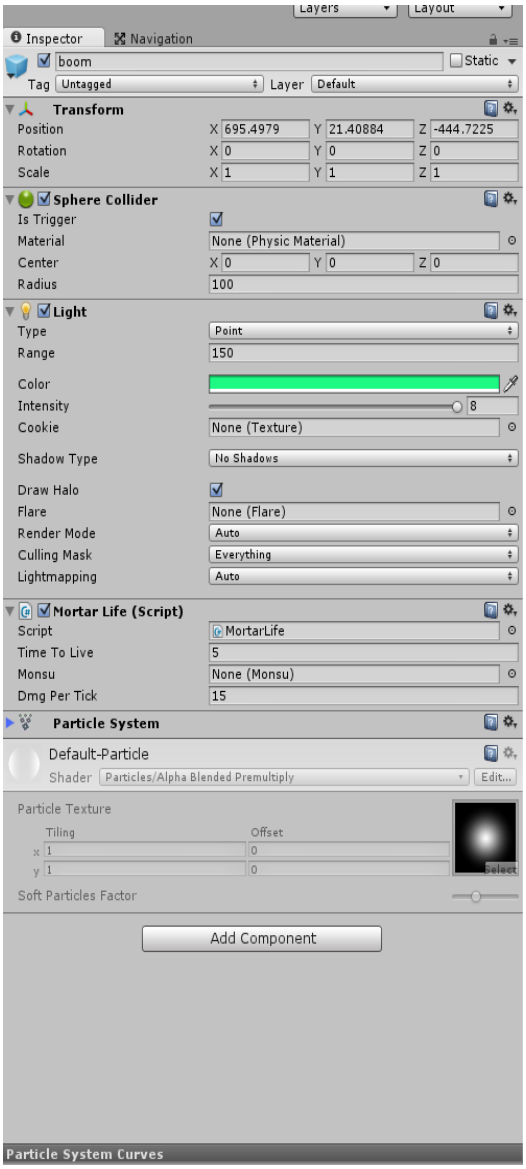


KUVA 2. Ruutukaappaus konsoli ikkunasta.

Game-ikkunan erikoisuutena on statistiikkalaatikko, josta on nähtävillä pelin optimointia helpottavia tietoja (kuva 3). Ruutu kertoo esimerkiksi pelin ruudunpäivitysnopeuden, käytetyn grafiikkamuistin määrän, sekä kuinka monta tekstuuria pelinäkymään piirretään kyseisellä hetkellä (Unity Manual: Rendering Statistics Window).



KUVA 3. Ruutukaappaus Game-ikkunan statistiikkaruudusta.



Inspector-ikkuna on yksi tärkeimmistä editorin ruuduista, sillä siitä nähdään yhdellä vilkaisulla tarpeellista tietoa valitusta objektista. Kuvassa 4 on valittuna gameobjecti nimeltään "boom". Inspector mahdollistaa objektin tarkan kohdentamisen kentällä, sekä kaikkien siinä olevien komponenttien hallinnoinnin. Monet komponenteista sisältävät niitä koskevia määritysvalikkoja, jotka saadaan auki inspectoriin. Skriptien osalta tässä ikkunassa näkyvät kaikki julkiset muuttujat, joten niiden arvoja voidaan muuttaa ilman koskemista koodiin. Myös koodissa tehdyt julkiset viittaukset erilaisiin objekteihin on mahdollista vetää suoraan tähän ikkunaan. Komponenttien perässä on sininen kysymysmerkillä varustettu kirjaikoni, jota painamalla Unity avaa selaimeen kyseistä komponenttia koskevan referenssisivun. (Unity Manual: Inspector.)

KUVA 4. Ruutukaappaus Inspector-ikkunasta.

4.3 Rakennuspalikat

Unity-projektit koostuvat monista pienistä palasista, joista rakentuvat pelin scenet. Sceneä on helpointa ajatella yksittäisenä kenttänä tai valikkona. Täysimittainen peli siis sisältää useita scenejä, joita on mahdollista muokata, sekä testata erillään editorin kautta, samalla pilkkoen latausaikoja osiin (Goldstone 2009, 29).

4.3.1 **Asset**

Assetilla voidaan tarkoittaa mitä tahansa, aina äänitiedostosta, 3D-malliin, animaatioon, tai pintamateriaaliin (Goldstone 2009, 15). Assettien luominen on tehty hyvin yksinkertaiseksi, sillä Unityn editorin ylälaidasta löytyy saman niminen valikko, jonka kautta saadaan luotua haluttu asset-pohja (Blackman 2013, 34). Asset-valikossa on myös kohdat import ja export. Import-toiminnolla voidaan omaan projektiin tuoda itse tehtyjä, kolmannen osapuolen, tai Unityssa valmiina olevia paketteja. Export puolestaan mahdollistaa omien pakettien luonnin nykyisestä projektista. (Smith & Queiroz 2013, 17-22.) Paketit ovatkin näppärä tapa nopeuttaa kehitystä, sillä niiden avulla voidaan tuoda valmiita assetteja vanhoista projekteista, tai esimerkiksi Unityn Asset storesta. Samalla vanhojen projektien aikana tehdyt valmiit osat eivät mene hukkaan, kun niistä voidaan tehdä paketti ja käyttää hyödyksi myöhemmin uudelleen.

4.3.2 **GameObject**

Gameobjectit ovat tärkein yksittäinen rakennuspalikka, sillä kaikki mitä kentällä näkyy, on vähintäänkin osa gameobjectia. Unity sisältää muutamia valmiita muotoja, joita voidaan käyttää, kuten neliö, ympyrä ja levy, sekä erilaisia valoja, mutta yleisin luotava gameobjekti on tyhjä. Tyhjään gameobjectiin voidaan liittää esimerkiksi 3D-malli, pintamateriaalit ja tarvittavat skriptit, jolloin siitä onkin muodostunut vaikka pelihahmo. Jokaisesta gameobjectista löytyy vähintään yksi komponentti, transform, joka määrittää sen koon, rotaation, sekä sijainnin kentällä perustuen Unityn koordinaatistoon (Goldstone 2009, 15). Gameobjectin mahdollisuudet ovatkin lähes rajattomat, sillä laajentamalla sen toiminnallisuutta erilaisilla komponenteilla, saadaan aikaan käytännössä lähes mitä tahansa peliin vaadittavaa (Blackman 2013, 34).

4.3.3 **Prefab**

Prefab on yksi assetin tyypeistä. Prefab voidaan luoda yhdestä, tai useammasta valmiista gameobjectista, se on siis periaatteessa tapa tallentaa gameobjecteja projektikansioon (Goldstone 2009, 16). Prefabin luomisen jälkeen siihen voidaan tehdä muutoksia projektikansion kautta, jolloin muutokset tulevat voimaan myös kentälle jo sijoitettuihin kyseisen prefabin ilmentymiin. Näin ollen se mahdollistaa usein käytetyn gameobjectin, esimerkiksi ammus tai puu, helpomman hallinnan, kun muutoksia ei tarvitse tehdä jokaiseen objektiin erikseen. Skriptien avulla voidaan myös luoda uusia gameobjecteja kentälle käyttäen valmista prefabia, jolloin esimerkiksi aseessa kiinni oleva skripti synnyttää ammusprefabin aina pelaajan painaessa hiiren nappia. (Unity Manual: Prefabs.)

4.3.4 **Component**

Unityn component-palikoita käytetään laajentamaan gameobjecteja. Aiemmin mainitsemani transformin lisäksi Unitysta löytyy komponentteja, joiden avulla gameobjectiin voidaan lisätä muun muassa fysiikkaominaisuuksia, törmäyksen tunnistimia tai ääniä (Blackman 2013, 34). Karkeasti voitaisiinkin sanoa komponenttien olevan rakennuspalikoita, joilla assetit kiinnitetään gameobjectiin tai avataan moottorin ominaisuuksia sen käyttöön. Esimerkkeinä annettakoon Mesh Renderer -komponentti, jolla gameobjectiin luodaan verkko johon taas voidaan kiinnittää materiaaliassetti, joka puolestaan muuttaa gameobjectin ulkonäköä. Toisena mainittakoon kamerakomponentti, jonka kiinnittämisen jälkeen pelin käynnistyessä pelinäkymä riippuu kyseisen gameobjectin sijainnista.

4.4 Ohjelmointi Unityssa

Puhuttaessa Unity-ohjelmoinnista, käytetään usein termiä skriptaaminen. Skriptit ovat lyhyesti sanoen kehittäjän itse tekemiä komponentteja, joiden avulla gameobjecteja ja muita assetteja voidaan kontrolloida huomattavasti laajemmin, verrattuna Unityn valmiisiin komponentteihin. Valmistajan kotisivujen kautta löytyy listaus, Unity scripting

reference, jossa on esillä Unityn omasta luokkakirjastosta löytyvät metodit. Näiden avulla kokematonkin kehittäjä pääsee hyvin alkuun skriptaamisessa. (Norton 2013, 9.) Skriptien avulla pelihahmo saadaan tottelemaan pelaajan syötteitä, voidaan luoda esimerkiksi graafisia elementtejä, tai käyttäytymismalleja gameobjecteille. Monikäyttöisyyden takia skriptaaminen onkin yksi aikaa vievimmistä osa-alueista pelin kehittämisessä.

Unityn mukana tulee skriptaamista varten erillinen ohjelma nimeltään Monodevelop. Tämä tarkoittaa sitä, että Unityn puolella luotu skriptikomponentti aukeaa automaattisesti Monodevelopmentissa, kun se avataan editointia varten. Ohjelmat on myös synkronoitu toimimaan yhdessä, joten Unity pitää molemmat ohjelmat ajantasalla kaikista lisätyistä, poistetuista, tai muutetuista skripteistä. (Norton 2013, 15.)

Skriptejä on mahdollista kirjoittaa tällä hetkellä kolmella eri ohjelmointikielellä, jotka ovat C# (lausutaan C-Sharp), Boo, sekä UnityScript (kutsutaan usein myös JavaScriptiksi, vaikka onkin Unityn muunnos siitä). Unityn kirjastot toimivat kielivalinnasta riippumatta ja esimerkit löytyvät jokaiselle näistä. (Unity Script Reference 2014.) Kielen valinta onkin oikeastaan kiinni kehittäjän omista mieltymyksistä, eikä ole loppuen lopuksi edes kovinkaan sitova, sillä eri kielilläkin kirjoitetut skriptit kykenevät kommunikoimaan toistensa kanssa tietyin rajoituksin (Unity Manual: Scripting 2012).

Henkilökohtainen valintani näistä kielistä on C#, sillä olen käyttänyt sitä pääsääntöisesti aina Unity-projekteissa ja sen syntaksi on hyvin lähellä muun muassa Javaa, jolla olen aloittanut ohjelmoinnin opiskelun. Javascript on myös varsin helppo kieli skriptaamiseen ja usein apua etsiessä vastaan tulee sillä toteutettuja ratkaisuja. Näin ollen monikielisyydestä on myös apua, koska vastaan voi tulla ongelmia, joihin ratkaisu löytyy vain yhdellä tuetuista kielistä. Boo on itselleni käytännössä täysin tuntematon, mutta ainakin syntaksiltaan se vaikuttaisi olevan jonkinlainen välimuoto kahdesta muusta vaihtoehdosta. Boo on hyvin vähän käytetty kieli, vain 0.44% kehittäjistä käyttää sitä, joten ongelmatilanteissa ratkaisujen löytäminen voisi olla vaikeampaa kuin kahden muun kohdalla. Tulevan Unityn viidennen version kohdalla myös dokumentaatio lopetetaan Boo -kielen osalta. (Unity Blog 2014.)

Unityn perusluokka on Monobehaviour, josta kaikki skriptit periytyvät (Unity Script Reference: MonoBehaviour). Tämä mahdollistaa Unityn sisäisten metodien ja luokkien käytön skripteissä.

Tällaisia ovat esimerkiksi metodit Awake ja Start, jotka ajetaan aina ensimmäisenä pelin latauskohdissa, joten niitä hyödynnetäänkin pääasiassa muuttujien alustamiseen ja tarvittavien viittauksien luontiin. Ajallinen järjestys kyseisten metodien kohdalla menee siten, että kaikki kentän Awake-metodit ajetaan satunnaisessa järjestyksessä ennen Start -metodeja. Mikäli siis muuttuja A tarvitsee alustamistaan varten muuttujan B olemassa olon, voidaan B alustaa Awake-metodissa ja A Start-metodissa. (Unity Script Reference: MonoBehaviour.Start.)

Pelin toiminnallisuuden luomisessa käytetyimpänä metodina voidaan pitää jokaisella ruudunpäivityksellä ajettavaa Update-metodia (Unity Script Reference: MonoBehaviour.Update). Tämän metodin sisässä suoritetaan suurin osa skriptin toiminnoista, joilla halutaan vaikuttaa reaaliaikaisesti gameobjectin toimintaan, kuten liikkumiseen, tai muihin tapahtumiin pelin aikana. Updatesta on olemassa myös muunnelma FixedUpdate (Unity Script Reference: MonoBehaviour.FixedUpdate), jonka ero normaaliversioon on siinä, että sitä kutsutaan ennalta määrätyin välein, jokaisen ruudunpäivityksen sijasta. Sitä käytetäänkin pääasiassa, kun halutaan vaikuttaa pelin fysiikkakomponentteihin kuten gameobjectin rigidbodyyn.

OnGUI-metodi on myös hyvin tarpeellinen, sillä sitä käytetään Unityn käyttöliittymäelementtien piirtämiseen ja niiden toimintojen hallintaan. OnGUI kutsutaan jokaisella ruudunpäivityksellä tai pelaajan käyttäessä käyttöliittymän elementtejä. (Unity Script Reference: MonoBehaviour.OnGUI.)

Yllä esitellyille metodeille yhteistä on se, että mikäli MonoBehavior poistetaan editorin kautta käytöstä, se estää kyseisten metodien suorittamisen (Unity Script Reference: MonoBehaviour).

5 PELIN TOTEUTUS

5.1 Suunnittelu

Pelin suunnittelu aloitettiin miettimällä, minkä genren edustajaa lähdettäisiin toteuttamaan. Aluksi idea oli toteuttaa avaruuteen sijoittuva strategiapeli, jossa pelaaja rakentaisi avaruusalusta ja sotisi sillä muita pelaajia vastaan, sekä suorittaisi yksinpelitehtäviä. Pidemmälle miettiessämme idea tuntui kuitenkin hyvin työläältä toteuttaa, kehitystiimin koostuessa vain kahdesta henkilöstä, joten lopulta päädyimme toteuttamaan avaruusteemaisen tornipuolustuspelin.

Tämän genren edustajia on nykypäivänä paljon, joten siihen vaadittavien ominaisuuksien pohtiminen onnistui helpoiten pelaamalla genren edustajia, kuten Ironhide game studion kehittämää Kingdom Rushia, sekä Hidden Path Entertainmentin Defence Grid peliä. Pelaamisen kautta saimme määritettyä omaan tuotokseemme perusrungon, jolla peli etenisi, sekä tärkeimmät mekaniikat, joita genren edustaja tarvitsee toimiakseen.

5.1.1 Pelin kulku

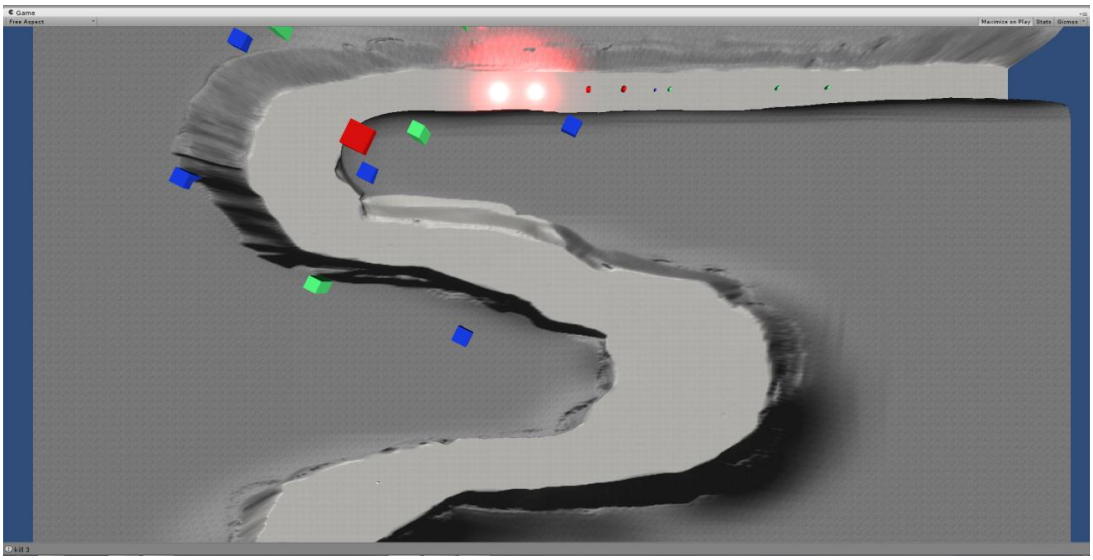
Peli etenee hyvin yksinkertaisesti alkuvalikosta ensin toiseen valikkoon, josta pelaaja voi valita kahdesta eri pelimuodosta haluamansa. Puolustusmoodissa peli etenee kentästä toiseen pelaajan onnistuessa puolustaa tukikohtaansa vaihtelevaa vihollisaaltojen määrää vastaan. Toisessa moodissa, eli selviytymisessä, vihollisaaltoja syntyy kentälle loputtomasti ja pelaajan tehtävänä on estää niiden pääsy kentän loppuun. Häviön tai puolustusmoodin kohdalla, viimeisen kentän voiton jälkeen, pelaaja ohjataan taas takaisin alkuvalikkoon.

Itse pelaaminen tapahtuu hiirellä, jota käyttäen pelaaja ostaa kentästä löytyville alustoille erilaisia puolustustorneja vihollisten harmiksi. Vihollisten toiminta on täysin automatisoitua ja ne etenevät raiteilla niille asetettua päämäärää kohti.

5.1.2 **Prototyypin rakentaminen**

Pelin ominaisuuksien määrittämisen jälkeen siihen alettiin toteuttaa tärkeimpiä komponentteja, jotka tämän pelin kohdalla omalta osaltani olivat tornien toiminta, vihollisten syntyminen kentälle, sekä niiden liikkuminen kentän läpi. Käytännössä tämä tarkoitti sitä, että testikentälle lisättiin esimerkiksi tyhjä objekti, johon toteutettiin skripti ja se alkoi synnyttää kentälle kuutioita, joilla simuloitiin vihollisia. Vähitellen skriptiin lisättiin uusia toimintoja niin kauan, että lopulta saatiin aikaan SpawnPoint-objekti, joka kuutioiden synnyttämisen lisäksi osasi myös laskea vihollisaallon koon ja tarkkailla, koska viholliset olivat hävinneet kentältä ja vasta sitten luoda uuden vihollisjoukkion. Kun objekti oli riittävän toimivassa kunnossa, siitä luotiin prefab projektikansioon ja siirryttiin kehittämään seuraavaa komponenttia.

Lopulta kun ylläkuvatun kaltainen kehitysprosessi oli tehty kaikille prototyyppiin vaadituille osioille, niistä kasattiin testikenttä, jossa kaikkia osioita voitiin testata ja jatkokehittää samanaikaisesti (kuva 5). Prototyypin pohjalta käynnistettiin uusi suunnitteluprosessi, jossa pelin olemassa olevia ominaisuuksia paranneltiin ja pohdittiin uusia komponentteja, kuten käyttöliittymää, ynnä muita tarvittavia lisäyksiä valmista versiota ajatellen. Prototyypin tekeminen helpottikin jatkosuunnittelua huomattavasti, sillä sen avulla pelin päämekaniikkoja pystyttiin testaamaan jo varsin kattavasti. Täten siinä havaitut puutteet, kuten vihollisten pysähteleminen vääriin paikkoihin saatiin korjattua varhaisessa kehitysvaiheessa.



KUVA 5. Ruutukauppaus pelin prototyyppi kentästä.

5.2 Ohjelmointi

Oma pääroolini pelin toteutuksessa oli pelin skriptien toteuttaminen. Pelin tämänhetkisessä versiossa on parisenkymmentä C#-skriptiä (ks. liite 1), joista käyn tarkemmin läpi sellaiset, jotka ovat kokonaisuudessaan omaa käsialaani. Käytännössä olen jossain määrin osallistunut muidenkin toteuttamiseen, ainakin testausmielessä, mutta niitä en käy tässä työssä läpi tarkemmin.

5.2.1 **Monsu.cs**

Monsu-skripti on kiinni jokaisessa pelikentälle luotavassa vihollisessa. Se pitää sisällään vihollisten osumapisteiden määrän, liikkumisnopeuden, sekä kuinka paljon vahinkoa vihollinen tekee pelaajan suojelemalle tukikohdalle. Skripti sisältää myös muutamia tarkistuksia, joiden tarkoituksena on estää etenkin kehityksen alkuvaiheessa esiintyneitä ongelmia vihollisten liikkumisessa. Esimerkiksi vihollisen törmäystunnistimen osuessa tukikohtaan, käynnistyy ajastin, jonka päästyä nollaan, Monsu-skripti hakee viholliselle uuden päämäärän, johon liikkua. Tällä tavoin estetään välillä esiintynyt ohjelmavirhe, jossa vihollinen saavutti tukikohdan, mutta ei lähtenyt sieltä enää liikkeelle. Liikkumisen varmistamisen lisäksi tämän skriptin toinen päätehtävä on tuhota kyseinen vihollinen kentältä, sen osumapisteiden loputtua. Tätä varten skriptissä on OnDeath-metodi, jota kutsuttaessa pelaajan resurssikertymää kasvatetaan vihollistyyppiä vastaava summa, lähetetään SpawnEnemy-skriptin laskuarvolle tieto tuhotusta viholisesta ja lopulta tuhotaan vihollisen gameobjecti kentältä.

5.2.2 **SpawnEnemy.cs**

SpawnEnemy on yksi pelin toiminnan kannalta oleellisimpia skriptejä, sillä sen päätehtävä on suorittaa vihollishahmojen instansioiminen kentälle. Tämä tapahtuu aalloittain ja yhteen aaltoon mahtuu tietty määrä vihollispisteitä. Eri vihollistyypit käyttävät vaihtelevan määrän kokonaisluvusta. Skripti kutsuu kuvassa 6 näkyvää selectMonsu-

metodia, joka arpoo jäljellä olevan kokonaislukumäärän mukaan seuraavaksi instansioitavan vihollisen case numeron spawnThis-muuttujaan. Mikäli jäljellä oleva määrä ei riitä tietyn vihollisen käyttämiseen, se jätetään arvonnan ulkopuolelle.

```
// Selects which monster will be spawned next, based on cap left.
public void selectMonsu(){

    int difference = enemyCap - enemySpawned;
    if(difference > 25){
        spawnThis = Random.Range(1,4);
    }
    else if(difference < 25 && difference > 10){
        spawnThis = Random.Range(1,3);
    }
    else if(difference < 10){
        spawnThis = 1;
    }

    spawnRdy = true;
    // Debug.Log("difference " + difference);
}
```

KUVA 6. Vihollisen valinnan suorittava metodi SpawnEnemy-skriptissä.

Arvonnan jälkeen skriptissä suoritetaan switch-case-rakenne, jossa toteutetaan spawnThis muuttujan mukainen valinta. Valinnat eroavat toisistaan vain vihollis-prefabin osalta, eli niissä luodaan uusi gameobjecti prefabin pohjalta, haetaan viittaus luodun vihollisen Nav mesh -agenttiin, jolle annetaan puolestaan liikkumiskohde sekä nopeus. (kuva 7.)

```
if (timeLeft <= 0.0f && letsSpawn) {

    // call function to select monster
    if(!spawnRdy && enemySpawned < enemyCap){
        selectMonsu();
    }
    else if(spawnRdy){
        // spawn
        switch(spawnThis){

        // goblin
        case 1:
            GameObject a = (GameObject)Instantiate(goblin, transform.position,transform.rotation);
            NavMeshAgent x = a.GetComponent<NavMeshAgent>();
            x.destination = destination.position;
            x.speed = a.GetComponent<Monsu>().speed;
            enemySpawned += 1;
            break;

        // soldier
        case 2:
            GameObject b = (GameObject)Instantiate(soldier, transform.position, transform.rotation);
            NavMeshAgent z = b.GetComponent<NavMeshAgent>();
            z.destination = destination.position;
            z.speed = b.GetComponent<Monsu>().speed;
            enemySpawned += 10;
            break;
```

KUVA 7. Switch Case rakenteen valinnat SpawnEnemy-skriptissä.

Vihollisten synnyttämisen lisäksi skriptissä on muuttujat aallon sisältämästä vihollisten kappalemäärästä, tuhotuista vihollisista, sekä selvinneistä vihollisista.. Näiden arvojen avulla skripti tietää milloin aallon kaikki viholliset ovat poistuneet kentältä ja uusi aalto voidaan synnyttää. Jokaisessa kentässä on tietty määrä aaltoja ja mikäli pelaaja onnistuu puolustamaan tukikohtaansa niiden ajan, SpawnEnemy käynnistää voittoikkunan.

5.2.3 **Ammusten skriptit**

Kaikissa puolustustornien synnyttämissä ammusobjekteissa on kiinni Projectile.cs-skripti. Sen tehtävänä on liikuttaa ammus Tower.cs-skriptin sille määrittämää maalia kohti. Jos maali katoaa matkalla, ammus tuhoaa itsensä. Liikkumiseen käytetään liukulukua stepSize, joka lasketaan jokaisella ruudunpäivityksellä uudestaan. Laskutoimitukseen käytetään ammukselle annettua nopeusarvoa, speed, sekä Unityn Time.deltaTime -arvoa. Lopputuloksena ammus liikkuu tietyn metrimäärän sekunnissa. (Unity Script Reference: Time.deltaTime.) Ammuksen saavuttaessa kohteensa skripti hakee kyseisen vihollisen Monsu-skriptin, jonka jälkeen se arpoo kyseisen ammuksen minimi ja maksimi vahinkoarvojen väliltä luvun ja kertoo viholliselle sen vahingoittuneen tämän verran. Mikäli vihollisen osumapisteet menevät osumasta riittävän alas, kutsuu vihollinen OnDeath-metodiaan oman Monsu-skriptinsä kautta.

Pelissä on erityyppisiä ammuksia, joiden erikoisefektit aktivoidaan niin ikään ammuksen saavutettua maalin. Aktivointi tapahtuu, mikäli ammuksen kyseistä ominaisuutta hallinnoiva boolean muuttuja on asetettu true arvoksi, jonka seurauksena kyseinen if-ehto suoritetaan. Erikoisammuksia on nykyisessä versiossa kaksi, joista toinen hidastaa vihollisen liikettä, vähentämällä vihollisen Nav mesh agentin nopeutta. Toinen luo osumakohtaan mortarBoom -nimisen gameobjectin, joka tekee vahinkoa sen läpi kulkeville vihollisille. Kaikki edellä mainitut tapahtumat ajetaan Projectile-skriptin Updat-metodissa (kuva 8).

```
// Update is called once per frame
void Update () {

    // destroy bullet if there is no destination
    if (destination == null) {
        Destroy(gameObject);
        return;
    }

    // moving towards enemy
    float stepSize = Time.deltaTime * speed;
    transform.position = Vector3.MoveTowards(transform.position, destination.position, stepSize);

    // when bullet hits its target
    if (transform.position.Equals(destination.position)) {

        // decrease monster hitpoints
        Monsu t = destination.GetComponent<Monsu>();
        dmgDone = GetRandom(minDmg,maxDmg);
        t.health = t.health - dmgDone;

        if(slowing){
            NavMeshAgent x = t.GetComponent<NavMeshAgent>();
            x.speed = t.GetComponent<Monsu>().speed - speedReduction;
        }

        if(mortarFire){
            Instantiate(mortarBoom, new Vector3(transform.position.x, transform.position.y, transform.position.z), Quaternion.identity);
        }

        // call monster's onDead if died
        if (t.health <= 0 && !t.killed){
            t.OnDeath();
        }
        // destroy bullet
        Destroy(gameObject);
    }
}
```

KUVA 8. Projectile skriptin Update-metodi.

Mortar-ammuksen synnyttämässä mortarBoom -objektissa on kiinni MortarLife-skripti, joka laskee tämän objektin olemassaoloaikaa, jonka täytyttyä objekti tuhotaan ruudulta. Samalla se tarkkailee törmääkö vihollinen alueen tunnistimeen ja mikäli näin tapahtuu se käynnistää Coroutine-metodin, joka poikkeaa normaalista metodista siten, ettei sitä tarvitse suorittaa yhden framen aikana. Tässä tapauksessa se siis aiheuttaa vahinkoa viholliselle useamman kuin yhden kerran. (kuva 9.)

```
public class MortarLife : MonoBehaviour {


    public float timeToLive = 4f;
    public Monsu monsu = null;
    public float dmgPerTick = 15f;

    //Coroutine for damage over time
    public IEnumerator MortarTick(){

        yield return new WaitForSeconds(1.0f);
        monsu.health -= dmgPerTick;

        yield return new WaitForSeconds(1.0f);
        monsu.health -= dmgPerTick;

        if(monsu.health <= 0 && monsu != null){
            if(!monsu.killed){
                monsu.OnDeath();
            }
        }
    }

    // Update is called once per frame
    void Update () {

        // Calculates when it is time to destroy mortarBoom
        timeToLive -= Time.deltaTime;

        if(timeToLive < 0){
            Destroy(gameObject);
        }
    }

    // Monster triggers courutine
    public void OnTriggerEnter(Collider other){
        if(other.gameObject.name.Contains("Monsu")){
            monsu = other.GetComponent<Monsu>();
            StartCoroutine(MortarTick());
        }
    }
}
```

KUVA 9. MortarLife-skriptin ajastin sekä Coroutine-metodi.

5.2.4 **Vihollisten päämäärät**

Vihollisilla on pelimuodosta riippuen jokin päämäärä: puolustuskentissä (defence) ne pyrkivät tuhoamaan pelaajan tukikohdan ja selviytymismoodissa (survival) niiden tehtävä on päästä pisteestä A pisteeseen B. Tukikohdassa, eli Castle-gameobjectissa on kiinnitettynä samanniminen skripti ja selviytymiskentissä Castlen korvaa Finish-objekti, johon on kiinnitetty SurvivalLine-skripti.

Castle-skriptin tehtävänä on tarkkailla omaa energia määräänsä sekä käynnistää vihollisten hyökkäysmetodi niiden tullessa Castle-objektin törmäystunnistimen sisälle. Vihollisen törmätessä tunnistimeen, Castle-skripti ottaa muuttujiinsa viittaukset vihollisen Monsu.cs-skriptiin, sekä vihollisen Nav mesh -agenttiin. Törmäys käynnistää myös kuvassa 8 esitellyn tyyppisen Coroutine-metodin, jonka aikana vihollinen vie tukikohdan energiaa. Viedyn energian suuruus määräytyy vihollisen robPerHit-muuttujan perusteella, joka löytyy Monsu-scriptin viittauksen kautta. Jos vihollinen ei ehdi tuhoutua coroutinen aikana, Castle asettaa vihollisen Nav mesh agentin päämääräksi SpawnPoint gameobjectin, johon on viitattu julkisella transform muuttujalla nimeltään back.

Energiamäärän seuraaminen tapahtuu Update-metodin sisällä. Skriptin energy -muuttujan mennessä nollaan tai sen alle, Castle-skripti luo viittauksen SpawnPoint-objektiin, käyttääkseen siinä kiinni olevan SpawnEnemy-skriptin ClearCurrent-metodia. Metodi tyhjentää tarvittavat muuttujat seuraavaa pelikertaa varten. Tämän tehtyään Castle antaa käskyn ladata Game Over -niminen kenttä. (kuva 10.)

```
// Update is called once per frame
void Update () {

    if(energy <= 0){

        // Load GameOver scene if castle is destroyed
        GameObject spawn = GameObject.Find("SpawnPoint");
        spawn.GetComponent<SpawnEnemy>().clearCurrent();
        Application.LoadLevel("Game Over");
    }
}
```

KUVA 10. Castle.cs Update-metodi.

SurvivalLine-skripti on ikään kuin yksinkertaistettu Castle.cs. Toiminnaltaan se on samantapainen, mutta vihollisen törmätessä sen tunnistimeen, skripti vain kasvattaa kokonaislukumuuttujaansa yhdellä ja käskee vihollisen tuhota itsensä. Tätä kokonaislukua se vertaa Update-metodissaan toiseen kokonaislukuun, joka pitää sisällään kentälle asetetun maksimiarvon, kuinka monta vihollista saa saavuttaa Finish-gameobjectin.

5.2.5 **GUI scriptit**

Unity GUI-luokka mahdollistaa erilaisten nappien ja laatikoiden tekemisen, joiden avulla pystytään toteuttamaan monimutkaisiakin valikkorakenteita tai tuomaan tarvittavia tietoja pelaajan näkyville. Itse olen toteuttanut tähän peliin pelimuodon valintavalikon sekä häviöruudun (kuva 11).

KUVA 11. Ruutukauppaukset pelimuodon valinta- ja häviöruuduista.

Pelimuodon valinta hoidetaan omassa kentässään, johon siirrytään päävalikon Play -napin kautta. Kenttä sisältää ainoastaan kameraobjektin, johon on kiinnitetty ModeMenu-skripti. Kuvassa 12 nähdään skriptin OnGUI-metodi. Metodin sisässä se piirtää sille annettujen paikkamääritysten mukaan ruudulle laatikon, joka sisältää kolme nappia. Näiden avulla pelaaja voi valita haluamansa pelimuodon, tai siirtyä takaisin päävalik-

koon. Lisäksi ruudulle piirretään taustalogo, mikäli sellainen on annettu viitteenä logo-tekstuurille.

```
void OnGUI() {
    GUI.skin = skin;

    Rect r = new Rect(Screen.width * (1 - widthPercent) / 2,
                      Screen.height * (1 - heightPercent) / 2,
                      Screen.width * widthPercent,
                      Screen.height * heightPercent);

    GUI.DrawTexture(new Rect(0, 0, Screen.width, Screen.height), background);

    if (logo != null) {
        Rect l = new Rect(Screen.width / 2 - 128, Screen.height / 9 - 18, logo.width / 2,logo.height / 2);

        GUI.DrawTexture(l, logo);
    }

    GUILayout.BeginArea(r);
    GUILayout.BeginVertical("box");

    if (GUILayout.Button("Normal Mode"))
        Application.LoadLevel("Defence1");

    if (GUILayout.Button("Survival Mode"))
        Application.LoadLevel("Survival1");

    if (GUILayout.Button("Back"))
        Application.LoadLevel("Start Menu");

    GUILayout.EndVertical();
    GUILayout.EndArea();

}
```

KUVA 12. ModeMenu.cs OnGUI-metodi.

Pelaajan hävitessä kentän, avautuu Game Over -kenttä, jonka sisältämä GameOverText-skripti piirtää ruudulle tekstuurin ja Update-metodin ajastimen täytyttyä, avaa pelin päävalikkoon (kuva 13).

```
void OnGUI () {
    GUI.DrawTexture(new Rect((Screen.width / 2) - 350, 150, 700, 400), endText);
}

// When time is full continue to start menu
void Update () {

        if(duration >= delay)
        {
            Application.LoadLevel(startFresh);
        }

        else
        {
            duration += Time.deltaTime;
        }
}
```

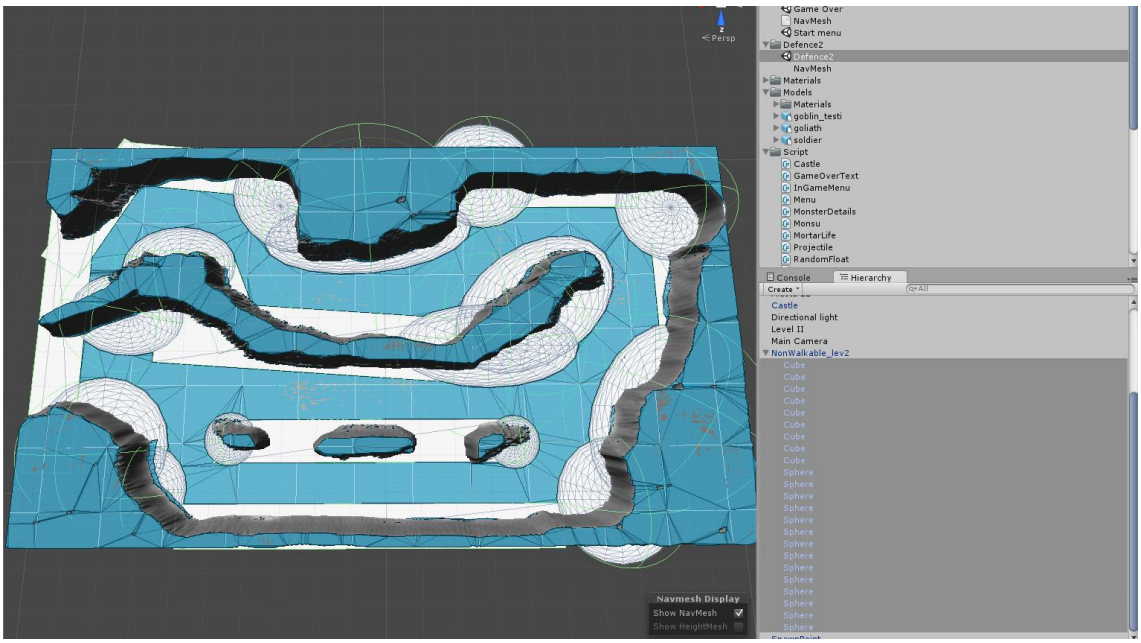
KUVA 13. GameOverText.cs OnGui- ja Update-metodit.

5.3 Reitinhaku

Pelin vihollisten navigointi on toteutettu kokonaisuudessaan Unityn sisäänrakennettua reitinhakutyökalua käyttäen. Työkalun käytön opettelu on pääpiirteissään hyvinkin helppoa ja sen avulla voidaan nopeasti luoda alueet, joita pitkin pelihahmot voivat liikkua ja vastavuoroisesti alueet, joihin ne eivät pääse.

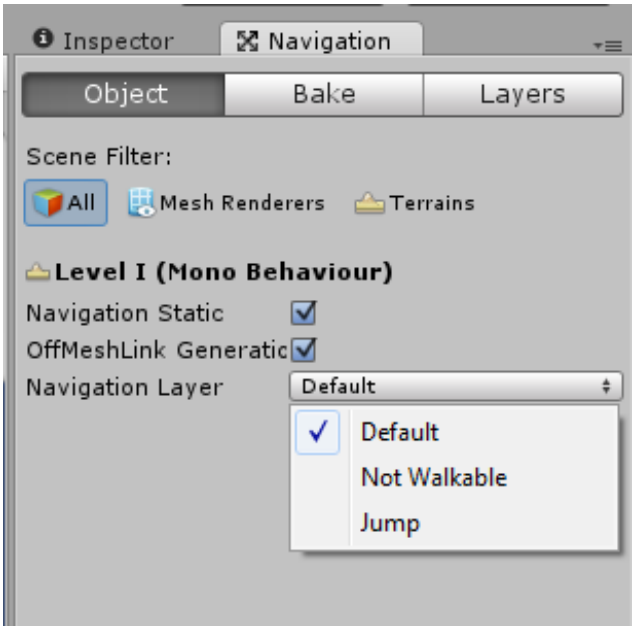
Tämän pelin kohdalla vihollisille tarvittiin reitti kahden gameobjectin välille, jota pitkin ne kulkevat. Tämä on toteutettu maalamalla koko terrain-osio kävelytäväksi maastoksi, vaikka todellisuudessa ne kulkevat vain pienellä osalla siitä. Kävelyalue rajattiin tekemällä tyhjä gameobjecti, jonka sisään tehtiin erilaisista mesh-objekteista koostuvat reunukset, joilla vihollisten liikkuminen on kiellettyä.

Kuvassa 14 nähdään Defence2 -kentän reitinhaku maalattuna. Valkoiset alueet ovat liikkumista rajoittavat reunukset, sinisillä alueilla liikkuminen on sallittua. Kun reitinhaku on saatu maalattua Bake-komentoa käyttäen, voidaan liikkumista rajoittavat reunukset poistaa kentältä.



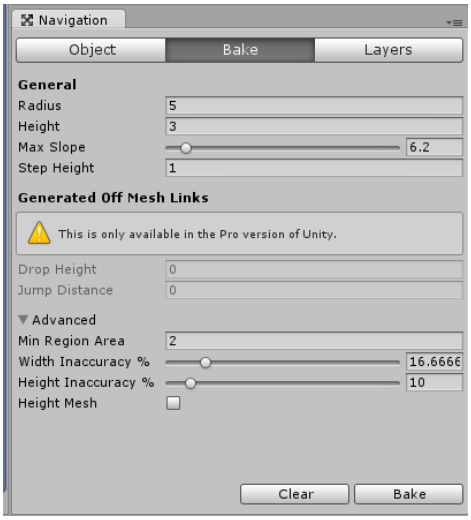
KUVA 14. Ruutukaappaus editorista, Defence2 kentän reitinhaku maalattuna.

Reitinhaun luominen aloitetaan merkitsemällä object-valikossa halutut osiot staattisiksi, jonka jälkeen valitaan, voidaanko kyseisellä alueella liikkua vai ei (kuva 15). Objektien valinta suoritetaan suoraan kenttänäkymästä tai hierarkiasta klikkaamalla.



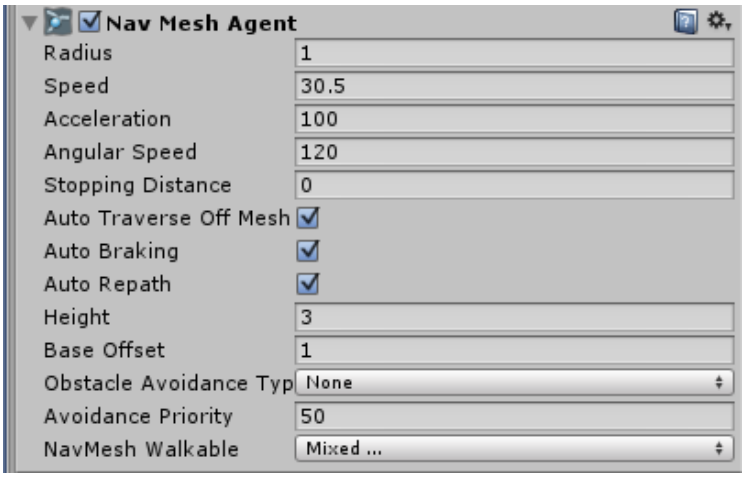
KUVA 15. Ruutukaappaus navigaatiotyökalun Object -valikosta.

Reitinhakualueelle voidaan antaa lisämäärityksiä Bake-valikossa (kuva 16), jossa liikkumisalueelle voidaan antaa muun muassa minimietäisyys reunoista, kuinka korkea liikuttavan tilan pitää olla, tai kuinka jyrkkiä luiskia pitkin hahmot voivat liikkua. Määritysten jälkeen kentän NavMesh luodaan Bake-nappia painamalla. (Unity Manual: Navigation Meshes.)



KUVA 16. Ruutukaappaus navigaatiotyökalun Bake -menusta.

Jotta reitinhaku toimisi, tulee sitä käyttäviin gameobjecteihin laittaa NavMesh Agent -niminen komponentti. Se saa kentästä luodun Nav mesh -pinnan kautta tarvittavat tiedot, joiden perusteella se laskee automaattisesti reitin pisteiden välillä. Agentti komponentille voidaan antaa monia määrityksiä (kuva 17), jotka vaikuttavat sen kiihtyvyyteen, nopeuteen, kääntyvyyteen, sekä kuinka nopeasti liike pysähtyy. Liikkeen lisäksi määritykset vaikuttavat muiden objektien väistämiseen, sekä reitinlaskemisen tarkkuuteen. (Unity Manual: NavMesh Agent.)



KUVA 17. Ruutukaappaus vihollisten Nav Mesh Agent -komponenttista.

Agentin käyttäytymistä voidaan muokata pelin aikana skriptien kautta. NavMeshAgent -luokka pitää sisällään monia muuttujia, joilla voidaan hienosäätää liikettä, tai vähentää reitin laskennasta aiheutuvaa laskentakuormaa. (Unity Script Reference: NavMeshAgent.) Tämän pelin kohdalla skriptejä on hyödynnetty muun muassa vihollisten vauhdin pienentämiseen, hidastustornin ampuessa niitä (kuva 18), sekä päämäärän vaihtamiseen vihollisen saavutettua tietyn pisteen kentällä (kuva19).

```
if(slowing){
    NavMeshAgent x = t.GetComponent<NavMeshAgent>();
    x.speed = t.GetComponent<Monsu>().speed - speedReduction;
}
```

KUVA 18. Nav mesh agentin nopeuden vähentäminen Projectile.cs skriptissä.

```
public void OnTriggerEnter (Collider other){

    // N connects to Colliders NavMesh Agent
    if(other.gameObject.name.Contains("Monsu")){
        n = other.GetComponent<NavMeshAgent>();
        monsu = other.GetComponent<Monsu>();
        roadToChoose = Random.Range(1,3);

        if(!toCastle && !monsu.done){
            if(roadToChoose < 1.5f && n != null){
                n.SetDestination(sideRoute.position);
            }
        }

        else if (toCastle && !monsu.done && n != null){
            n.SetDestination(castle.position);
        }

        if(monsu.done && n != null){
            n.SetDestination(backSideRoute.position);
        }
    }
}
```

KUVA 19. Nav mesh agentin päämäärään muuttaminen ChangeRoute.cs skriptissä.

5.4 Ongelmat toteutuksessa

Pelinkehityksessä ongelmat ovat melko yleisiä, koska käytännössä toteutus on etenkin koodillisesti jatkuvaa ongelmaratkaisua, kuinka joku tosielämän asia voidaan luoda peliin. Run the Gauntletin kohdalla suurimmat ongelmakohdat kohdistuivat tornien toimintalogiikkaan ja reitinhaun toimintaan.

Tornien kohdalla eniten vaikeuksia aiheutti ammusten lentoradan laskeminen, jotta ne osuisivat vihollisobjekteihin. Useiden eri kokeilujen jälkeen ratkaisu löytyi Unityn dokumentaatiota selailemalla ja lopulta ammuksen liike vaati vain muutaman rivin koodia toimiakseen. Toinen torneja koskeva ongelma oli niiden ostovalikon aukeaminen, sillä jostain syystä kaikkien kentällä näkyvien alustojen ostovalikot aukesivat päällekkäin ja torni syntyi alustalle, joka oli asetettu kentälle ensimmäisenä, vaikka pelaaja olisikin klikannut eri alustaa. Ratkaisuna oli, aina pelaajan klikatessa alustaa, hakea kyseisen alustan nimi ja asettaa sen sijainti syntymiskohdaksi. Käyttöliittymän aukeaminen saatiin puolestaan lisäämällä skriptiin muutama boolean muuttuja, joiden avulla käyttöliittymä pakotetaan sulkeutumaan pelaajan valitessa ostettava torni.

Edellä mainitut pois lukien, skriptaamiseen liittyneet ongelmat olivat hyvin pieniä ja niiden selvittäminen onnistui Unityn Debug -luokkaa käyttämällä. Sen avulla skriptien toiminnasta voidaan lähettää viestejä konsoliin, joiden avulla voidaan seurata toimivatko esimerkiksi muuttujat tai metodit halutulla tavalla.

Reitinhaun kanssa koetut ongelmat liittyivät pitkälti pelimoottorin toimintaan ja ne selvisivät uudelleen maalamalla kentän reittipohja löyhemmäksi, tai muuttamalla reitinhakuagentin määrityksiä. Välillä ongelmat olivat taasen käyttäjäpohjaisia eli esimerkiksi vihollisia synnyttävä objekti oli jäänyt liian korkealle kentän pinnasta, eikä agentti tästä syystä päässyt kosketuksiin reittipohjan kanssa.

6 JULKAISU

Alkuperäinen tarkoitus oli julkaista Run the Gauntlet jossain digitaalisessa jakelupalvelussa, mutta peli ei tämän opinnäytetyön puitteissa saavuttanut vielä laadullisesti tarvittavaa tasoa, jotta se olisi ollut kannattavaa. Siispä tässä luvussa käsitellään teoreettisesti, minkä kanavan kautta julkaisu olisi voinut tapahtua.

Pc-pelien kohdalla digitaalinen myynti on ajanut fyysisen kappaleiden myynnin ohi viime vuosina ja joidenkin arvioiden mukaan jopa 92% Pc-pelien myynnistä tapahtuisi digitaalisten jakelukanavien kautta (Sacco 2014). Digitaalisen jakelun kautta pelinkehittäjä voi myös toimia samalla julkaisijana, jolloin välikäsiä on vähemmän ja näin ollen kehittäjälle jää suurempi osa pelin mahdollisista tuotoista. Samalla kehittäjällä on vapaammat kädet toimia, koska taustalla ei ole toimintaa rahoittavaa julkaisijaa, joka voisi ohjata kehittäjän toimintaa. (Suomen Pelinkehittäjät ry 2010, 7-8.)

Julkaisupaikan suhteen valinnanvaraa on hyvin paljon. Internetistä löytyy hyvin erikokoisia toimijoita ja aina vaihtoehtona on jakaa peliä omien sivujensa kautta. Valven Steamin ja Electronic Artsin Originin kokoisiin kauppapaikkoihin oman luomuksen saaminen vaatii kuitenkin huomattavasti enemmän, verrattuna vaikkapa indiekehittäjille suunnattuihin sivustoihin.

Pienemmät sivustot ovatkin hyvä vaihtoehto, sillä niiden ympärille on monesti rakentunut kehittäjien ja pelaajien yhteisö. Yhteisön kautta voi omasta pelistään saada arvokasta palautetta, vaikkei siitä heti hittituotetta syntyisikään. Palautteen kautta peliään voi kehittää eteenpäin ja julkaista uuden version, joka saattaakin iskeä jo kultasuoneen. Julkaisun voi tehdä myös useassa paikassa samaan aikaan, jolloin saatu palaute, tai tuotto voivat moninkertaistua, verrattuna yhden julkaisukanavan käyttämiseen.

6.1 Steam Greenlight

Greenlight on Valven Steam-palveluun luoma väylä, johon kehittäjät voivat laittaa pelistään ruutukauppauksia, videoita ja muuta infoa. Steamin käyttäjät pystyvät antamaan palveluun laitetuille peleille ääniä ja antaa kehittäjälle palautetta pelistä. Mikäli peli saa tarpeeksi huomiota osakseen, sen on mahdollista saada julkaisulupa Steam-palveluun. Julkaisun kannalta tärkein rajoitus on, että pelin tulee toimia ainakin Windows-alustalla. Greenlightissa ei ole minkäänlaista aikarajaa, joten vaikka peliä ei hyväksyttäisi julkaistavaksi, se häviää palvelusta vasta kehittäjän itse poistaessa sen. Pelin lähettäminen palveluun vaatii Steam-tilin, johon on rekisteröity vähintään yksi peli, sekä kertaluontoisen, noin 80 euron suuruisen lähetysmaksun. Maksun jälkeen kehittäjä voi lisätä Greenlightiin niin monta peliä kuin haluaa. (Steam Greenlight, 2014.)

6.2 Desura

Desura on julkaisupaikka, johon voidaan lähettää valmiita, tai kehityksen loppusuoralla olevia pelejä. Pelin saaminen palveluun edellyttää, että se läpäisee palvelun ylläpitäjän testauksen ja on laadullisesti riittävällä tasolla. Apua kehitykseen voi hakea Desuran yhteydessä toimivasta MODDB-yhteisöstä. Ennen julkaisua kehittäjä määrittää pelilleen haluamansa hinnan ja lisää tarvittavat tiedot, sekä vähintään yhden trailerin sivustolle. Kun peli hyväksytään palveluun, ylläpitävä taho testaa lopullisen version ja on kehittäjään yhteydessä vaadituista korjauksista ennen lopullista julkaisua. Julkaisun jälkeen kehittäjä voi tehdä peliin päivityksiä, joiden julkaisu käy läpi saman laadunvalvonta prosessin. (Desura: How to 2014.)

6.3 IndieGameStand

IndieGameStand on sivusto, johon kehittäjä voi rekisteröityä ja lähettää hakemuksen, jonka perusteella ylläpitäjä hyväksyy pelin palveluun. Palveluun päästyään kehittäjälle avautuu kehittäjän työkalut, joiden avulla pelin tiedot ja tarvittavat tiedostot ladataan sivustolle. Tämän jälkeen peli lähetetään kauppapaikalle, jonka jälkeen ylläpitävä taho tarkastaa kaiken olevan kunnossa ja lähettää kehittäjälle ilmoituksen, kun peli on laitettu myyntiin. (IndieGameStand, Youtube 2013.) Palveluun voidaan lisätä myös testausvaiheessa olevia tuotoksia ja ainoat rajoitukset ovat, että pelin tulee toimia joko Windows- tai Mac-alustalla. Mahdolliset tuotot jaetaan siten, että kehittäjä saa 75% ja sivusto 25% (IndieGameStand 2014).

7 POHDINTA

Tämän opinnäytetyön tavoitteena oli antaa tilaajalle yleiskuva pelinkehitysprosessista sekä luoda mahdollinen ponnahduslauta pelien pariin, jonka avulla yritys saisi jatkossa laajennettua toimintaansa myös pelien pariin. Tavoite täyttyi ainakin osittain, sillä kehityksen aikana molemmat osapuolet saivat kosketuksen siihen, mitä näinkin pienen pelin toteutus ajallisesti vaatii ja minkälaisia taitoja toimivan pelin tekeminen tarvitsee. Siitä poikiiko tämä projekti uusia, ei ainakaan tällä hetkellä ole varmuutta, mutta pienellä parantelulla pelistä saa aikaan varsin hyvän referenssityön jatkoa ajattelen.

Tarkoituksena puolestaan oli toteuttaa Pc:llä pelattava peli Unity-pelimoottorin avulla, jota olisi jatkossa helppo laajentaa päivityksillä. Työn toinen tarkoitus oli tutkia, miten Pc-peli saadaan julkaistua digitaalisesti. Tämän hetkinen versio Run the Gauntletista on käytännössä pelattava, mutta kehityksen aikana siitä karsiutui pois ominaisuuksia, jotka toisivat peliin paljon lisää interaktiota pelaajan kanssa. Myöskin grafiikat ja äänipuoli vaatisivat lisäyksiä ja parantelua julkaisua varten. Peli on kuitenkin rungoltaan varsin toimiva, joten siihen on helppo lisätä uusia kenttiä, torneja tai muita ominaisuuksia. Vaikka pelin julkaisu siirtyikin tulevaisuuteen, on tässä työssä läpikäydyt kanavat varsin otollisia keinoja saada Run the Gauntlet julkaistua tilaajan niin halutessa.

Kokonaisuudessaan tämä työ osoittaa ainakin sen, että pienelläkin ryhmällä on mahdollista nykypäivänä toteuttaa pelejä. Saatavilla on monia työkaluja, joiden avulla pelien toteuttaminen on varsin helppoa, kunhan ryhmästä löytyy jonkinlaista kokemusta ohjelmoinnista, grafiikan teosta ja äänien toteuttamisesta. Tässäkin työssä käytetty Unity on pelimoottorina varsin pätevä ja aloittelijaystävällinen, mutta tarvittaessa sillä voidaan luoda hyvinkin näyttäviä tuotoksia. Myöskään raha ei ole enää nykypäivänä este pelien toteuttamiselle, sillä tarjolla on täysin ilmaisia vaihtoehtoja kehitystyökaluiksi, eikä julkaisunkaan mahdollistamiseksi tarvita mittavia resursseja.

LÄHTEET

Blackman, S. 2013. Beginning 3D Game Development with Unity 4: All-in-One, Multiplatform Development. New York: Apress

Desura: How to. 2014. Bad Juju Games. Luettu 12.11.2014.
http://www.desura.com/groups/desura/howto

Doran, J. 2013. Mastering UDK Development HOTSHOTS.
Birmingham: Packt Publishing

GameMaker Documentation. 2014. Luettu 24.9.2014.
http://docs.yoyogames.com/

GameMaker: Studio. 2014. Luettu 24.9.2014.
http://www.yoyogames.com/studio

Goldstone, W. 2009. Unity Game Development Essentials.
Birmingham: Packt Publishing

Hiltunen, K. Latva, S. Kaleva, P. 2013. Peliteollisuus kehityspolkuna. Tekes.
Luettu 23.8.2014. http://www.tekes.fi/Julkaisut/peliteollisuus_kehityspolku.pdf

IndieGameStand: How to list your indie game in our store in ~5Minutes. Youtube 2013.
Katsottu 12.11.2014.
http://www.youtube.com/watch?v=ZSWuJnoPgOk#t=254

IndieGameStand. 2014. Indie Developers. Luettu 12.11.2014.
http://indiegamestand.uservoice.com/knowledgebase/topics/9437-indie-developers

Neogames: Alan toimijat. 2014. Luettu 17.11.2014.
http://www.neogames.fi/tietoa-toimialasta/alan-toimijat/

Neogames: Tietoa toimialasta. 2014. Luettu 17.11.2014.
http://www.neogames.fi/tietoa-toimialasta/

Norton, T. 2013. Learning C# by Developing Games with Unity3D Beginner's Guide.
Birmingham: Packt Publishing.

Rigney, R. 2010. The Origins of Angry Birds. PCWorld. Luettu 23.8.2014.
http://www.pcworld.com/article/206831/the_origins_of_angry_birds.html

Sacco, D. 2014. Digital downloads account for 92% of PC game sales. PCR.
Luettu 10.11.2014.
http://www.pcr-online.biz/news/read/digital-sales-make-up-92-of-global-game-revenues/034551

Smith, M., Queiroz, C. 2013. Unity 4.x Cookbook.
Birmingham: Packt Publishing

Steam Greenlight. 2014. Luettu 12.11.2014
http://steamcommunity.com/workshop/about/?appid=765&section=faq

Suomen Pelinkehittäjät ry. 2010. Suomen pelitoimialan strategia 2010-2015. Neogames. Luettu 11.11.2014.
http://www.neogames.fi/wp-content/uploads/2013/05/Pelistrategia-2010-2015.pdf

Unity3D. Luettu 12.9.2014.
http://unity3d.com/unity

Unity Blog. 2014. Documentation, Unity Scripting languages and You. Aleksandr. Luettu 10.9.2014
http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/

Unity: FAQ. 2014. Unity3D. Luettu 14.11.2014.
http://unity3d.com/unity/faq

Unity Learn: The new UI. 2014. Unity3D. Katsottu 29. 9.2014.
http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/the-new-ui

Unity Manual: Inspector. 2014. Unity3D. Luettu 2.10.2014.
http://docs.unity3d.com/Manual/Inspector.html

Unity Manual: Navigation Meshes. 2014. Unity3D. Luettu 2.11.2014.
http://docs.unity3d.com/Manual/Navmeshes.html

Unity Manual: NavMesh Agent. 2014. Unity3D. Luettu 3.11.2014.
http://docs.unity3d.com/Manual/class-NavMeshAgent.html

Unity Manual: Prefabs. 2014. Unity3D. Luettu 3.10.2014.
http://docs.unity3d.com/Manual/Prefabs.html

Unity Manual: Rendering Statistics Window. 2014. Unity3D. Luettu 2.10.2014.
http://docs.unity3d.com/Manual/RenderingStatistics.html

Unity Manual: Scripting. 2012. Unity3D. Luettu 29.9.2014.
http://docs.unity3d.com/401/Documentation/Manual/Scripting.html

Unity: Public Relations. 2014. Unity3D. Luettu 12.9.2014.
http://unity3d.com/public-relations

Unity Script Reference. 2014. Unity3D. Luettu 9.9.2014.
http://docs.unity3d.com/ScriptReference/

Unity Script Reference: MonoBehaviour. 2014. Unity3D. Luettu 25.10.2014.
http://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Unity Script Reference: MonoBehaviour.FixedUpdate. 2014. Unity3D. Luettu 25.10.2014.
http://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html

Unity Script Reference: MonoBehaviour.OnGUI. 2014. Unity3D. Luettu 25.10.2014.
http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnGUI.html

Unity Script Reference: MonoBehaviour.Start. 2014. Unity3D. Luettu 25.10.2014.
http://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html

Unity Script Reference: MonoBehaviour.Update. 2014. Unity3D. Luettu 25.10.2014.
http://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

Unity Script Reference: NavMeshAgent. 2014. Unity3D. Luettu 3.11.2014.
http://docs.unity3d.com/ScriptReference/NavMeshAgent.html

Unity Script Reference: Time.deltaTime. 2014.Unity3D. Luettu 24.10.2014.
http://docs.unity3d.com/ScriptReference/Time-deltaTime.html

Unity: Store. 2014. Unity3D. Luettu 14.11.2014.
https://store.unity3d.com/

Unreal Engine. 2014. Luettu 15.9.2014.
https://www.unrealengine.com/what-is-unreal-engine-4

Unreal Engine Documentation: Programming Basics. 2014. Luettu 23.9.2014.
https://docs.unrealengine.com/latest/INT/Programming/Basics/index.html

Unreal Engine Showcase. 2014. Luettu 15.9.2014.
https://www.unrealengine.com/showcase

Unreal Script Reference. 2012. Luettu 23.9.2014.
http://udn.epicgames.com/Three/UnrealScriptReference.html

Ward, J. 2008. What is a Game Engine?. GameCareerGuide. Luettu 25.8.2014.
http://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=1

LIITTEET

Liite 1. Luokkakaavio Run the Gauntletista

Laatinut Karlo Tuominen, RefleKT Media 2014